Plotting and defining functions in R

We use functions in two ways in this project. The first is (hopefully) more familiar, the second is how we will proceed in the rest of the class. We will get to that in Section .

Our first analogy will be functions as they are used in describing population distributions. R has a number of built in populations, we will show how to define new ones in Section .

The function `dnorm` computes the density of a normal distribution. We know that a normal has two paramters – a mean, μ, and a standard deviation σ. As such, `dnorm` has two additional argument, here called `mean` and `sd`:

```
> dnorm(0)

[1] 0.3989423

> dnorm(0, mean = 1)

[1] 0.2419707

> dnorm(0, mean = 1, sd = 2)

[1] 0.1760327
```

In the first case, the *default* values of `mean=0` and `sd=1` are used. The second specifies a value for `mean` (leaving `sd` its default). The third uses no default values.

The variables are *vectorized* which is important, especially for the first or `x` value. By this we mean that if the `x` value stores several numbers, the function will compute several values. For example:

```
> x <- c(0, 1, 2, 3)
> dnorm(x)

[1] 0.398942280 0.241970725 0.053990967 0.004431848
```

Among the other built in R functions for distributions are `dt`, for the t-distribution; `dunif`, for the uniform distribution; and `dexp`, for the exponentail distribution,

To plot a function on a graph can be done with the `curve` function [1]

For example,

[1] `curve` does nothing more than find a bunch of x values, apply the function to generate y values then make a plot of the x and y values by plotting them and connecting with line segments (dot-to-dot).

```
> curve(dnorm(x), from = -3, to = 3)
```

Or

```
> curve(dnorm(x, mean = 10, sd = 10), from = -20, to = 40)
```

Question 0.1. Make a plot of the exponential distribution `dexp` from 1 to 100. Afterwards, replot using a sensible range for the x values. What values did you think sensible?

The `curve` function will make a new plot, as above, when we specify values of `from` and `to`. If there is a current plot, we can add a function with the argument `add=TRUE`. For example,

```
> x <- rnorm(100, mean = 100, sd = 30)
> hist(x, main = "Histogram with density", probability = TRUE)
> curve(dnorm(x, mean = 100, sd = 30), add = TRUE)
```

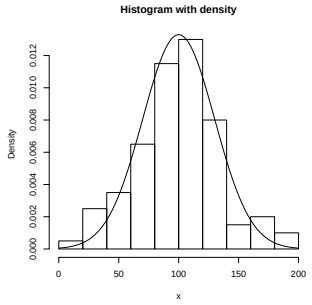

Figure 1: Histogram with density added by `curve`. To make this work, the histogram is created so its total area adds to 1. The `curve` function has the argument `add=TRUE` to get the function to appear on the current plot.

Question 0.2. Create these values for x:

```
> x <- rexp(100)
```

Make a histogram, then layer on the density (`dexp`).

Question 0.3. The argument for `curve`, `lwd=2`, makes lines twice as wide, `lty=2` specifies a different line type, and the argument `col="blue"` draws a line in blue. (You can of course you different widths and colors.) These are useful when you have more than one curve on the same graphic. Explore these values by redoing the above graphic.

The `function` keyword

To create your own function is not to difficult. The only trick – and it isn't necessary, but is good form – is to make your function accept vectorized arguments.

The keys to defining a function are:

Declaring it is a function using the function keyword
All function definitions begin with the keyword `function`.

Define the variables you want A function can have 0, 1 or many variables. Many of the functions we use have 1 or more parameters in addition to the x value(s). For example the arguments of `dnorm` include `x`, `mean` and `sd`.

Compute the return value, then make sure it is returned A function consists of a bunch of steps that turn the input into the output value. The value returned (or outputted by the function) is the last value created. Of then the `return` function is used to make this unambiguous.

For example, to define a function to do $f(x) = x^2$, we have

```
> f <- function(x) {
+     x^2
+ }
```

And to use it:

```
> a <- 1:3
> f(a)

[1] 1 4 9
```

The keyword `function` on the right hand side says make a new fuction. We assign this to the variable `f`, but it could be any valid variable. The `(x)` part contains the argument. In the call (`f(a)`) we show that the value in the call need not be named `x`, as the position is used to match arguments. In this example, the only command executed is also the last which here squares the number and returns it. From the call of the function, we can see it is vectorized.

The *arguments* to the function are specified within the parantheses. There can be 0, 1, or more than 1 specified.

The braces make a command block. The return value of the last command executed is returned. In this case, the command `x^2` is the last one, so this is returned.

Question 0.4. Make a function to find $f(x) = 5 + 6x$. Plot over the interval $[0, 10]$.

Question 0.5. Make a function to find $f(x) = e^x/(1+e^x)$. What is the value at 1? Plot over the interval $[-10, 10]$. (You use `exp(x)`, not `e^x`)

Arguments

The value of `x` is passed into the function call through function arguments. In this example, there is only the one, but we see that most of our functions have a parameter family. These values may be specified too by the user, but the programmer must alert the function.

We may want to have paramters with defaults. For example, a triangle distribution has the basic shape of an equilateral triangle from -1 to 1 with a peak at 1. (This has area 1). To program this in, we have

```
> f <- function(x) {
+     if (x < -1)
+         0
+     else if (x < 0)
+         1 + x
+     else if (x < 1)
+         1 - x
+     else 0
+ }
```

We used `if` and `else`, but hopefully the logic is clear,

To specify a function with parameters, we essentially relate a function with a different mean and scale according to

$$g(x;c,h) = \frac{1}{h} f(\frac{x-c}{h})$$

The bit $\frac{x-c}{h}$ is like a z score, the outside $1/h$ keeps the area equal to 1. So to make triangle distribution with mean `mean` and scale `h` we could have:

```
> g <- function(x, mean = 0, h = 1) (1/h) * f((x - mean)/h)
> g(0)

[1] 1

> g(0, mean = 1)
```

```
[1] 0

> g(0, mean = 1, h = 2)

[1] 0.25
```

When an argument is specified as `name=value`, the `value` is the default value for the argument and need not be specified. In the above, we call the first value by "position" and the others by "name." Calling by name is clearer to read, but calling by position involves less typing. Often it wins out.

However we are not done with our triangle example, we can not graph this function with `curve` – it is not vectorized. To do so, we get a bit fancy with our definition of f so that it is vectorized. [2]

```
> f <- function(x) {
+     1 - sign(x) * pmax(pmin(x, 1), -1)
+ }
```

With this defined, we can make some plots:

```
> curve(g(x), from = -2, to = 2, lty = 1)
> curve(g(x, mean = 1), add = TRUE, lty = 2)
> curve(g(x, mean = -1, h = 2), add = TRUE, lty = 3)
```

Question 0.6. Make a function in R to do the following

$$f(x;a,b) = axe^{-bx}.$$

Make a plot of $f(x;2,2)$ over the interval 0 to 5. What is the maximum value? Where does it occur?

Question 0.7. Make a function in R to do the following

$$f(t;a,k,t_0) = a(1-e^{k(t-t_0)}), \quad t > t_0$$

Use `x` for the main argument (t above).

Plot with $a = 10$, $k = 2$ and $t_0 = 1$. Plot over the interval $[1,10]$. Describe the shape.

[2] You won't need to be this fancy unless you go off on your own. The `pmax` and `pmin` functions are vectorized max and min functions, and `sign` just gives a 1 if positive and -1 if negative (basically $x/|x|$).

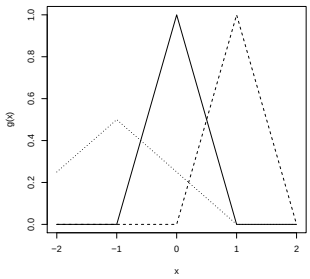

Figure 2: Three triangle distributions differing by choices of parameters.

Adjusting parameters

Most functions fall in *families*. A family of functions shares common featurs – bell shaped, long tailed, certain decay, S-shaped, ... – but there will be *parameters* that can be tuned to fit the scale of the problem at hand.

An example, which you have encountered before, comes from the idea of a probability distribution. Recall, for a continuous distribution, the area under the curve between a and b represents a probability that a random number is in $[a,b]$. For example, the normal curve provides such a curve. You remember that the normal curve is in a family with parameters μ and σ. We now these as the mean and standard deviation, but more generally they represent the *center* and *scale* of the data. For the normal curve, you used the fact that finding the z-score ($z = (x-\mu)/\sigma$) allows one table to be used for all normals. You may also recall that for the t distribution, this wasn't the case, so you used a restricted table.

One of the keys to fitting models is to find the right parameters. This will be done with mathematical optimization routines, but these often require good starting values. These are often found by "eye-balling" the data and guessing initial values, or by fitting a function to the data and adjusting parameters until it "looks right."

We begin with a demonstration where we adjust parameters of a probability distribution to match the data, as represented by a histogram. This may not work – I didn't get a chance to test – but to try we first download and install a package for R: [3]

[3] The `traitr` package requires some other software to be installed on the computer you work at and so may not work for you.

```
> install.packages("traitr")
```

If that is successful, you can download the following file:

```
> f <- "http://www.math.csi.cuny.edu/verzani/classes/MTH703/Data/explore-params.R"
> source(f)
```

This creates two demos where you can adjust sliders to try and fit a density to a histogram. The normal distribution with two parameters – the mean and standard deviation; and the exponential distribution with its one parameter – in this

case the mean. To do this, you eyeball the mean and spread then adjust the parameter values. You can check by clicking the help button, and play again by clicking the OK button. The two demos are run with either

```
> do_norm()
> do_exp()
```

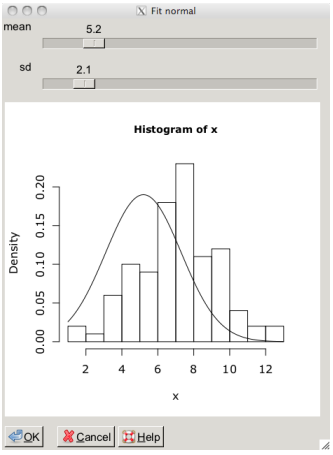

Figure 3: GUI for exploring how a density can be adjusted to "fit" the histogram. The density is a theoretical model for the sample data summarized by the histogram.

Functions for fitting trends

A more common usage in this class for functions is to use them to describe trends in data. The basic idea employed in this class is our response variable depends on the predictor(s). The response can be regarded as

$$\text{response} = \text{model} + \text{random error}$$

The model is a function of the predictor values that specifies the mean value of the response. The "random error" part specifies this is a statistical model, where we don't assume we can model the exact value of the response, rather we model the mean value (using the model part) and describe the random bit using a distribution.

This is exactly the assumptions behind simple linear regression where the model is written

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i,$$

where the the model is linear: ($\beta_0 + \beta_1 x$ is the equation of a line with slope β_1 and intercept β_0) and we make the assumption that the ε_i values are independent and normally distributed.

To explore how the parameters affect the model, you can try the demo `do_lm`.

This class introduces numerous other functions into the mix beyond the linear model.

We will use a data set on predator prey relationships by Sinclair et al. 2003, *Patterns of Predation in a diverse predator-prey system* Nature 425, 288-290 (18 September 2003).

You can get it with:

```
> f <- "http://www.math.csi.cuny.edu/verzani/classes/MTH703/Data/prey-mortality.csv"
> pm <- read.csv(f)
> g <- "http://www.math.csi.cuny.edu/verzani/classes/MTH703/Data/prey-range.csv"
> pr <- read.csv(g)
```

The `pm` data is from the prey's perspective, the `pr` from the predators. Run the `str` function to find out more about the new data sets.

We fit a few models by guessing parameters.

Linear model

We warm up with a linear model. Our function might look like

$$f(x;a,b) = a + bx$$

which is coded into R as

```
> f <- function(x, a = 0, b = 0) a + b * x
```

For a predator, the maximum size of its prey is related to the predator's size. We plot the data and try a linear fit:

```
> plot(max.prey ~ wt, pr)
> curve(f(x, a = 0, b = 500/150), add = TRUE)
```

Question 0.8. Adjust the parameters to see if you can get a better fit.

Question 0.9. As there is a wide range of weights and clumping near 0, it might be better to compare on a log-log scale. With that in mind, redo the plot with:

```
> plot(log(max.prey) ~ log(wt), pr)
```

Then add a curve until you get one that seems to fit best.

Hockey stick model

The hockey stick model is one with a constant value for some period and a linearly increasing or decreasing value before or after. The function is marked by a) a slope and intercept when it is increasing, b) a point when it changes. The latter is actually easier to specify in terms of the maximum (or minimum value). Here is an R function to do one type of hockey stick.

```
> hs <- function(x, a = 0, b = 0, sat = 0) {
+     pmin(sat, a + b * x)
+ }
```

For the predator mortality rate data in `pm` we have that this increase as there are more predators – if everyone is hunting for you, you are less likely to live to a ripe old age. We plot the data and draw on a possible fit with these commands:

```
> plot(pred.mortality.perc ~ no.predators, pm)
> curve(hs(x, a = 0, b = 30, sat = 100), add = TRUE)
```

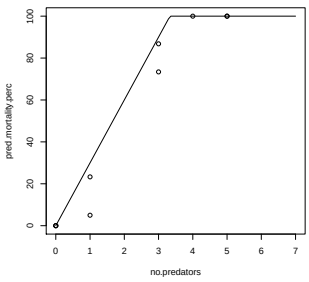

Figure 4: Data with hockey stick fit. Above a certain number of predators, a prey is almost certain to be killed by one.

Question 0.10. Plot the above. Adjust the parameters to better fit the data. In order to do so, you need to figure out what `a`, `b` and `sat` represent. What do you find?

Exponential decay

The relationship between number of predators and body weight is basically that larger animals have fewer predators. But how to quantify that? We plot the data, and then fit two models – a hockey stick, and a modified exponential. Here are the function families. (We redo `hs`, can you tell why?)

```
> hs <- function(x, a = 0, b = 0, sat = 0) pmax(sat, a + b * x)
> e <- function(x, a = 0, k = 1, x0 = 0, sat = 0) {
+     sat + (a - sat) * exp(-k * (x - x0))
+ }
```

These commands start the fit.

```
> plot(no.predators ~ wt, pm)
> curve(hs(x, a = 7, b = -1/200, sat = 0), add = TRUE)
> curve(e(x, a = 7, k = 1/200, x0 = 0, sat = 0), add = TRUE)
```

Question 0.11. Adjust the parameters above to find the best fit. What is better the exponential or the hockey stick? Why?